Challenger I-P Memory Map (BASIC-in-ROM Configuration)

0000 - 00FF	Page Zero
0100 - 01FF	Stack
*0130	NMI Vector
*01C0	IRQ Vector
0200 - 0221	BASIC Flags & Vectors
*0203	LOAD Flag
*0205	SAVE Flag
*0218	Input Vector
*021A	Output Vector
*021C	Control C Check Vector
*021E	Load Vector
*0220	Save Vector
0222 - 02FA	Unused
0300 end of RAM	BASIC Workspace
A000 - BFFF	BASIC-in-ROM
D000 - D3FF	Video RAM
DF00	Polled Keyboard
F000 - F001	ACIA Serial Cassette Port
F800 - FBFF	ROM
FC00 - FCFF	ROM - Floppy Bootstrap
FD00 - FDFF	ROM - Polled Keyboard Input Routine
FE00 - FEFF	ROM - 65V Monitor
FF00 - FFFF	ROM - BASIC Support
*FFFA	NMI Vector
*FFFC	Reset Vector
*FFFE	IRQ Vector

MEMORY LOCATIONS CONTAINING THINGS OF INTEREST

000B,C	Address of USR routine
000D	Number of extra nulls to be inserted after carriage return
000E	Number of characters since last carriage return
000F	Terminal width (for auto CRLF)
0010	Terminal width for comma spaced columns
0013-5A	Input buffer
005F	String variable being processed flag (?)
0061	?
0064	CTRL O flag (hi bit on = suppress printing)
0065	sometimes contains $68 (??)
0079,7A	Pointer to initial null of BASIC program workspace
007B,7C	Pointer to beginning of BASIC variable storage space
007D,7E	Pointer to beginning of BASIC array storage space
007F,80	Pointer to end of array space/beginning of free memory
0081,82	Pointer to end of string space/top of free memory
0085,86	Pointer to top of memory allowed to be used by BASIC
0087,88	Current line number
0089,8A	Sometimes next line number (?)
008F,90	DATA pointer
0095,96	This is where ADOB leaves address of the variable it found
0097,98	Address of variable to be assigned value by OUTVAR (AFC1)
00AA,AB	Points to pointer of next BASIC line after LIST
00AD,AE	The contents of this pair is printed in decimal by B962
00AE,AF	This is where INVAR (AE05) leaves its argument
00D1-D7	Clobbered by OSI Extended Monitor disassembler;kills BASIC
00E0-E6	Apparently unused page zero space
00E8-FF	Apparently unused (by BASIC) page zero space
00FB	ROM monitor load flag
00FC	ROM monitor contents of current memory location
00FE,FF	Address of current ROM monitor memory location
0130	NMI routine
01C0	IRQ routine (can be overwritten by stack being used by BASIC)
0200	Current screen cursor is at D700 + (0200);initialized to (FFE0)
0201	Save character to be printed
0202	Temp storage used by CRT driver
0203	LOAD flag ($80=LOAD from tape)
0205	SAVE flag (0= not SAVE mode)
0206	Time delay for slowing down CRT driver
0207-0E	Variable execution block-code for screen scroll-not reuseable
0212	CTRL C flag (not 0=ignore CTRL C)(reset by RUN)
0213-16	Polled keyboard temporary storage and counter
A000-37	BASIC initial word jump table (in token order; add 1 to each ad
A038-65	BASIC non-initial word jumps (real entry addresses)
A084-163	BASIC keywords in ASCII;hi bit set as delimiter;in token order
A164-86	Error messages with null delimiter
BE4E	"Written by" message

VERY USEFUL BASIC ROUTINES

00BC Works its way through a line of BASIC (or whatever C3,C4 points to) and gets the next char each time it is called. It will be pointing to the end of your USR statement if you call it from the USR; you can then use it to get stuff after X=USR(Y)--and BASIC will never be the wiser! BC leaves carry set if character is numeric.

00C2 Entry to the BC routine without incrementing C3,C4 before getting the character. Thus it gets the current character.

A477 Call this routine and then jump to A5C2 and you'll be RUNning the current BASIC program--starting from machine language!

A925 Call this from a USR statement and you will be doing an INPUT statement--but BASIC will not echo the characters you type in--including the CRLF at the end. This gives you a real BASIC INPUT statement that doesn't screw up your nice graphics by scrolling the screen one line! You must set loc 64 to $80 (set the CTRL O flag) before this all works. Do an LSR $64 to clear the flag to normal if you want BASIC print statements to work again.

AAC1 Like AAAD but no type mismatch check.

AAAD One you've been waiting for. This gets a 16 bit argument from the current BASIC line position (yes, like right after the ")" of your USR statement!), evaluating whatever expressions it finds, and leaves it where a call to AE05 will find it and put it in AE,AF! (Use AC01 to find a comma and then call AAAD again to get another value!)

ABF5-AC0C This series of routines (actually of entry points to one routi uses the BC routine to check for various delimiters. If you disassem the ROM here, it demonstrates a classic use of the 2C opcode as a combination NOP and immediate load, depending on where you jump in. ABFB checks for ")"; ABFE for "("; AC01 for ","; AC03 for whatever character you leave in A when you call it. ABF5 checks for "(", calls AAC1 to get a value, then checks for ")". (Thoughts of a BASIC statement X=USR(Y)(Z) should be jumping into your head about now.)

AD0B This routine uses the BC routine to find the name of the variable that's next in the BASIC line, and puts the address of the variable in locs 95,96. It also leaves the address in A,Y. If you store A in 97 and Y in 98, you can call OUTVAR (AFC1) to store whatever 16 bit value you put in A and Y into that BASIC variable.

B3AE This is like AAC1, but gives an error if the value is greater than 255_{10}. (Used by the POKE routine to keep you from putting a too-big number in memory.)

B962 Prints the decimal value of whatever 16 bit number is in AD,AE at the current cursor location on the screen, with normal BASIC checks for line length (does auto CRLF if line is too long) etc.

MISCELLANEOUS BASIC ROM ROUTINES

These notes do not claim to be complete or even error-free. They are only my hastily scribbled comments on those routines I happened to come across in my looking at BASIC.

0000 Warmstart (4C 74 A2)

0003 Message printer (A8C3)

00A1 Genl purp JMP instr; put target addr in A2,A3

00BC Get next char in BASIC line

00C2 Get current char in B line

A1A1 Look back thru stack ???

A212 Check for OM and stack overflow

A24C "OM" error

A24E Error; caller sets X-reg to error code

A274 Warmstart entry

A357 Input and fill buffer; put null at end

A386 Input from FFEB

A399 Toggle CTRL O flag

A432 Find BASIC line whose # is in 11,12; put addr of ptr of that line in AA,AB

A477 Point C3,C4 to 0301;reset str and array ptrs;reset stack to (1)FC;put 0301 in 8F,90;0 in 8C;0 in 61; 68 in C5 (?)

A491 Clear stack;0in 8C and 61

A5C2 Top of main BASIC exec loop

A5FC Entry to BASIC execute loop

A5FF Do line of BASIC

A629 Jmp FFF1 for CTRL C

A636 CTRL C entry point

A67B Set null count at D0 (?)

A77F Get dec # from buffer; put value in 11,12

A866 Put null at end of buffer; CRLF;nulls

A86C CRLF w/ nulls from 0D

A8C3 Msg printer; A,Y point to msg, which ends w/ null

A8E0 Output " "

A8E3 Output"?"

A8E5 Output char in A; update 0E; check line length

A925 Input routine less clear CTRL O

A946 Output "? ";jump to A357

AAC1 Like AAAD w no TM err check

AAAD Get 16 bit arg from BASIC line; AE05 will put value in AE,AF; does TM err check.

ABA0 Put 0in 5F;get char;goto B887 if numeric ???

ABD8 16 bit complement using AE05/AFC1 ?

ABF5 Checks for "(", calls AAC1,checks for ")"

ABFB SN err if next char not ")"

ABFE SN err if next char not "("

AC01 SN err if next char not ","

AC03 SN err ifnext not what's in A

AC0C SN err printer

AD0B Get var name from BASIC line; put addr of var in 95,96 and A,Y

AD53 Expects var name in 93,94; finds addr of var and put in 95,96 and A,Y; 0 in 61

AE05 INVAR puts 15 bit signed value in AE,AF

AE85 BS error

AE88 FC error

AFC1 OUTVAR 0 in 5F;(A)in AE; (Y) in AF; then to ?

B0AE Msg printer (A8C3)

B3AE Put 8 bit arg from line in AE,AF

B3F3 (BA,BB) to C3,C4

B4D0 Arith to normalize FP arg??

B887 Check for +,-,$,#,.,E... long!

B95A Prints current line #

B962 Prints contents of AD,AE (as dec)

BD11 Coldstart

BEE4 UART input routine (S1883 chip at FB0X)

BEF3 UART output routine

BEFE UART initialization

BF07 ACIA input (6850 chip at FC0X-like CII-4P)

BF15 ACIA output routine

BF22 ACIA initialization

ROM BASIC NOTES

Here is what we know so far of the structure of OSI ROM BASIC Version 1.0 rev 3.2.

A good place to start exploring is the warmstart entry at A274. (All addresses are hex unless otherwise noted.) BASIC can also be warmstarted by a jump to loc 0000--where the system puts 4C/74/A2 at coldstart. At this point, BASIC is looking at the keyboard, waiting for immediate mode commands or BASIC instructions with line numbers to be entered.

See the warmstart flowchart. BASIC first clears the CTRL O flag (LSR $64 clears the flag--the hi bit of loc 64) to allow printing, invokes the message printer (loc 0003 is a jump to the printer at A8C3) by the standard convention of pointing A,Y (lo,hi) at the message (ASCII in RAM or ROM--with last character of a null--that delimiter tells the rpinter routine to return) and prints "OK crlf". (The OK is stored at A192,3) Now the "fill the input buffer" routine is called. This routine (at A357) inputs (thruugh FFEB, from either keyboard or ACIA, depending of the load flag loc 0203, bit 7) characters, keeps a count of them, stores them in the imnput buffer at loc 13-5A, handles "backspace", @, CTRL O, and when it sees a CR, calls A866 to put a null instead of a CR in the buffer, and print a CRLF with extra nulls from OD. (Nulls are put in the output stream after CRLF if needed for a slow device by puttin the number of nulls desided in loc OD.) There is also a flowchart for A357, a main system routine.

There exists a vital routine callable at 00BC (the code for which is copied at coldstart from BCEE-BD05 in ROM) that puts the next character in the current line being worked on in the accumulator. (The current character may be had in A by calling 00C2 instead of BC. The BC routine also sets the carry flag if the character being passed is numeric, for the information of the calling program. The address of the current character is in loc C3,C4--the address portion of an LDA instruction. Everybody uses BC to find out what's up next. C3,C4 is constantly be changed by the users of the BC routine, in addition to being incremented by BC each time it is called.

MISCELLANEOUS NOTES ON BASIC

Try answering "A" to C/W/M?--A for author.

All final quotation marks are optional unless ambiguity would result. For example, PRINT "JIM works fine, but INPUT "NAME ; A$ does not.

If you want to embed commas in a line you are typing in response to an INPUT statement, begin the line with quotation marks. This will also let you enter a line with leading blanks. The same thing also lets you put commas or leading blanks in DATA statements. The closing quotes are, of course, optional (unless ambiguity would result).

A colon after any response you type to an INPUT statement ends what the INPUT sees, but lets you make remarks on the screen. For example, if inresponse to INPUT A$ you type JIM:WILLIAMS <RET> the screen will show what you typed, but A$ will contain only "JIM".

Although it is not documented, the statement ON X GOSUB nn,mm,pp,... works just fine--just the same as an ON X GOTO, but calling subroutines.

Recovery from coldstart is possible if you answer "MEMORY SIZE?" with a number instead of <RET>. (Once you hit RETURN, BASIC fills the memory with test bytes until it doesn't get them back to see how much memory there is. That means your program is completely and irrevocably overwritten.) The easiest way is to go into the ROM monitor before you coldstart and find and copy the contents of locations 007B,7C and 0301,02. Then coldstart, entering your memory size (i.e. 4096 for a 4K machine, etc.) and after BASIC comes up, go back to the monitor and replace 7B,7C (the end of program/beginning of variables pointer) and 0301,02 (the pointer from the first BASIC statement to the second, which will be set to zeros by coldstarting--though the rest of the program is still there). If you have already coldstarted, look for the first zero byte after loc 0305, and put an address one higher than that zero in 0301,02 (low order byte first; the contents of 0302 will be 03 always, unless you have hand-manufactured a very unusual BASIC program.) The program will now list, but will wipe itself out if you try to run it. (Variables will overwrite the beginning of the program.) List the program, immediately use the monitor to find the contents of 00AA,AB, and put those contents into 007B,7C. Everything should then be back to normal. (In fact, immediately after listing any line, locations AA,AB will contain the address of the pointer of the next BASIC statement--or of the beginning of variable space if the last line of the program is listed.)

Long BASIC lines produce auto carriage return/line feeds when listed. When saving on tape, this causes the last part of the line to be lost. By setting the "TERMINAL WIDTH" to longer than any BASIC line with a POKE 15,255, the damaging carriage return will be avoided.

If you have some program in the machine, but want to look at a program on a tape without writing over the program already there, the following "VIEW" program will be useful. It is absolutely relocatable, so may be put anywhere in memory; it reads tapes and writes only on the screen.
20,07,BF,20,EE,FF,D0,F8,F0,F6. Starting address is first byte.
This won't work on 1P's; the ACIA is in the wrong place.

HOW TO READ A LINE OF MICROSOFT

We are going to be talking about a lot of numbers in the next few paragraphs. It would probably be easier to visualize if you had the numbers in front of you on your system. If you have an OSI system, I would suggest that you turn it on and enter this program:

```
10 B=0:A$="P"
20 FORX=769TO830:?PEEK(X);:NEXT
```

Now run the program before we go any further.

If you have run the program, you are now looking at the entire text and variable table for a small program. OSI MICROSOFT reserves the first three pages of memory for housekeeping duties so the text actually begins at location 769 - the first location that you displayed. The first line of the program should be coded:

```
16 3 10 0 66 171 48 58 65 36 171 34 0
```

The first two bytes, 16 3, are the location of the next line of program. The next two bytes are the number of the current line (10 0) and the end of the line is marked by a 0. (0's are often used as markers in MICROSOFT as they occur infrequently in text storage.)

All of the commands, what MICROSOFT calls "reserved words", are encoded in MICROSOFT codes. The arithemitic operators (,-, *,/,and) are also considered commands and encoded. The 171's appearing in the line are " " statements. (I have included a list of the MICROSOFT codes with this data sheet.)

MICROSOFT uses ASCII to store print statements, remarks, variable names, and, strangely enough, all numbers that appear in the text. All line numbers in GOTO statements, all arithmetic values, all variable values, and all values in IF statements are stored in ASCII. Miscellaneous characters such as brackets and " marks are normally stored in ASCII.

The only thing that does not seem to have a hard and fast rule are REM and DATA statements. Those two commands may be found either in ASCII or code and seem to work as well either way. There does not appear to be any discernable pattern to the choice of method of storage.

The ASCII representation of numbers is significant. It explains why statements using variable names normally execute faster than statements using the numerical values for the operation. BASIC has to convert the ASCII numbers to BCD for storage and to HEX for arithemetic operations before they can be used. Variable values are already processed and ready in a table and can be looked up faster than they can be converted.

Here, the BC routine is being used to work through the ASCII in the input buffer as it is being tokenized. C3,C4 is set to point at the input buffer. If the first character in the buffer is numeric, the buffer must contain a numbered line of BASIC source, so we go to A295 to do the "tokenize and store in BASIC workspace, updating necessary pointers" job on the input buffer. If the first character is not numeric, we call A3A6 to tokenize the line in the buffer and put it back in the buffer. Then we jump to A5F6, the main entry to the execute BASIC statements loop.

When a program is RUN (from the beginning), A5F6, in executing the immediate mode command RUN, jumps to the RUN routine at A477, which does the following: 1)points C3,C4 to the contents of 79,7A (the beginning of BASIC workspace)(0301); 2)resets the string pointer at 81,82 to the top of memory as recorded in 85,86; 3) resets the array pointer to the end of the BASIC program (also known as the beginning of BASIC single variable space) as kept in 7B,7C. (This pointer at 7B,7C is constantly updated during BASIC editing and program entry.); 4) the 6502 stack pointer is reset to (01)FC; 5) a 00 is stored in locs 8C and 61 (why?); 6) a $68 is stored in loc 65 (why?). Returning from A477, we jump to A5C2, the top of the "do the next line of BASIC" loop. See the "Main BASIC execution loop" flowchart.

In the main BASIC loop, at A5C2, we first do a CTRL C check, and stop, printing "BREAK IN LINE"(contents of 87,88) before returning to warmstart if we find CTRL C. If not, we check to see if the next character in whatever line we're working on is a null (the beginning of another BASIC line). If it isn't, it had at least better be a ":" to indicate multiple statements per line, or we go to the syntax error printer, and back to warmstart. If we have a null, the hi byte of the pointer after it will contain a 00 if we are at the end of the program, so if we find that, we stop. Otherwise, it's on to the next line of BASIC, first storing the number of this new line in 87,88, and then incrementing C3,C4 past the pointer and line number. The next sequential instruction in ROM is A5FC, and we continue executing BASIC statements.

A5FC is the main entry point to the "run the BASIC program"loop. See its flowchart. It calls BC and checks for a null--and exits to warmstart if it finds that trivial case. Otherwise it calls A5FF to do the dirty work of executing a BASIC statement before looping

A5FF calls BC and checks to see if the first character is greater than $80. If not, it is not a token, so we must be doing a LET statement with an implied LET. In this case, we go to A7B9, which calls AD0B, a very important subroutine that finds the name of the variable the LET will assign into, finds its address in variable storage space, puts that address in 95,96, and also returns with the address in A,Y. A7B9 then checks for an "=" (everybody, of course, using BC to find the next character) (if no "=", then syntax error), calls important routine AAC1, the "evaluate an expressio routine (with no checking for TM error) and somehow stores the output value of AAC1 into the address AD0B left. Done with the statement, we return to A5F6, which loops back to the top at A5C2. (There will be a short quiz on these addresses at the end of the period

If A5FF finds a token at the beginning of the line, it first verifies that it is an initial word token (i.e., less than $9C) then does an ASL, TAY to multiply the token value by 2 to get an offest for the initial word jump table at A000. (Note on tokens: Tokens are functionally divided into initial words like FOR, RUN,POKE, and other non-initial words like THEN,=,SQR. There is a subroutine to handle each initial word, and the addresses of those routines are stored in a table at A000, two bytes per routine, since it take two bytes for an address. The addresses are stored in the order of the token numbers; that is, the first address is for token 80, the next address (A002,A003) is for token 81, etc. Initial tokens go up through 9B. For non-initial tokens, some (like SQR) are complex enough to require their own subroutines, while others (like =) do not. Tokens 9C through AC require no subroutines; AD through C3 do. The first 28 tokens (the initial word ones) take 28*2 bytes in the table, so the non initial tokens get the addresses starting after the first 56 bytes of the table, namely at A038. (The 28 and 56 are decimal.) Ignoring the hi bit of an initial token and multiplying it by 2 gives the address in the table of the routine for that token.) (If you think that's hard to follow, it's even rougher to infer from a disassembled dump of the ROMs!) Anyway, A5FF now has the address of the subroutine that will do the operation of the BASIC keyword that started the line. It pushes this address onto the stack, calls BC (for the convenience of the next routine) and an RTS does the actual jump to the needed routine. Again: the address of the routine to do the desired BASIC operation for an inntial word is pushed onto the stack--like the return address is

for a JSR--and then an RTS makes the processor jump there. This all happens around A60D. (Small detail:A5FF JMP's to BC; subroutine BC's RTS is what actually pops the address off the stack and "returns" there.) (Another detail: Since the PC is incremented by one after popping the return address from the stack, the addresses in the initial word part of the jump table are all 1 lower than the routines' actual entry addresses.)

The other, non-initial tokens are dealt with within the initial word routines. The routines to service the non-initial tokens that are complex enough to need them are called by the old ASL,TAY trick. (The ASL is at A627; the TAY at AC55) That offset in the Y-register is added to an invented base address of 9FDE to find the routine's address in the jump table.(9FDE + 2*(AD with hi bit ignored)=A038, the address of the jump for the routine for token AD.)(Phew!) This jump is not a stack trick; so the addresses in the jump table for non-initial tokens are correct as they stand. (They don't have to have 1 added to get the real address.) The 9FDE+Y stuff is around AC56.

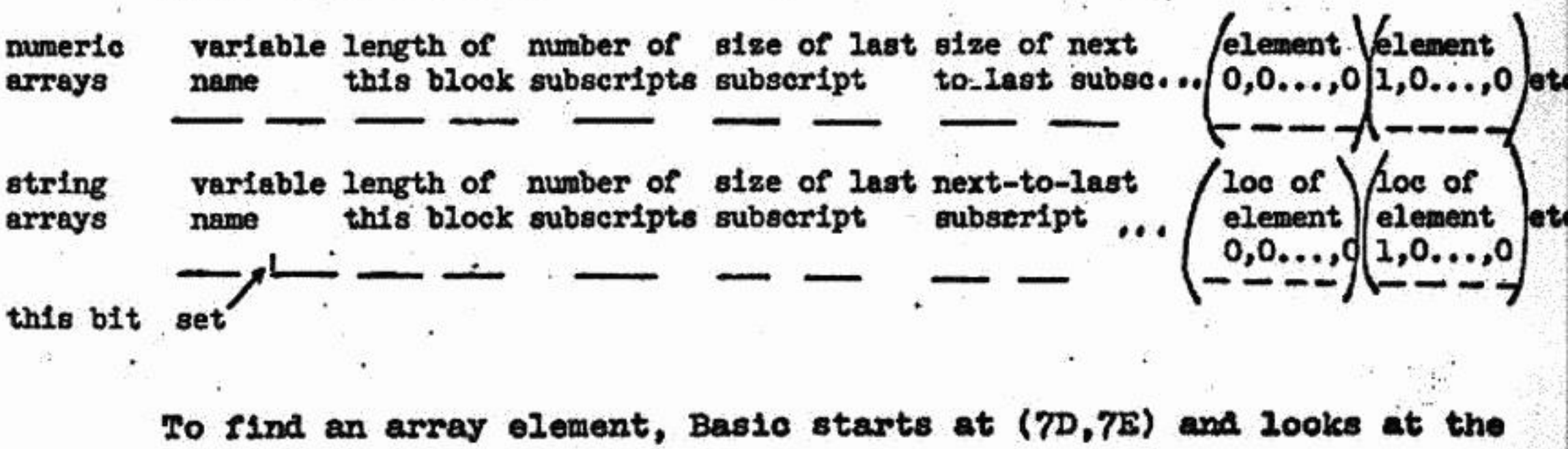
Program to look at binary representations of numbers in memory

```
10 INPUT M
20 P=PEEK(123)+256*PEEK(124)
30 P=P+2
40 FOR J=0 TO 3
50 N=PEEK(P+J)
60 GOSUB 200
70 PRINT "  ";
80 NEXT
90 PRINT
100 GOTO 10
200 FOR I=0 TO 7
210 B=N AND 2^(7-I)
220 IF B THEN PRINT "1";:GOTO 240
230 PRINT "0";
240 NEXT
250 RETURN
```

(Yes, lines 210 and 220 are correct.)

The program waits for you to input a number, then prints the binary representation of it, and then waits for another number.

Arrays are stored in assorted length blocks from (7D,7E) to (7F,80) as follows:

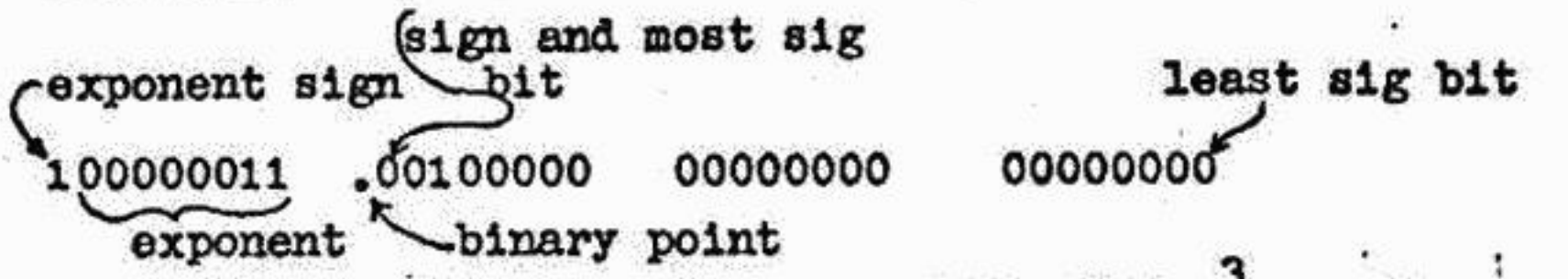


To find an array element, Basic starts at (7D,7E) and looks at the name, then skips to the name in the next block (that's why we have that 3rd byte) etc until a match is found, then skips 4 bytes per element until it finds the element it wants. (If it's a string, we have the length and location of the string, not the actual string. This table is over at (7F,80).

Strings are actually stored starting at the top of memory (as indicated by (85,86)). Modifying the contents of 85 and 86 (or having answered a number less than the actual memory size to "MEMORY SIZE?" at coldstart) will keep the strings from wiping out any other programs or data you may want to tuck safely away in the top of RAM. BASIC uses this space at the top of the memory with no regard for saving space or reusing space unless it runs out of space. It keeps a pointer to the next (working from top to bottom) free space in (81,82), putting any strings it needs (array or not) there and updating the pointer until it runs out of room. (I.e., (81,82)=(7F,80)) To keep from creaming the array tables (the first thing it would run into), BASIC calls a "garbage collection" routine that tries to shuffle the strings around to the top of the memory and reclaim unused space. Unfortunately, there seems to be a bug in the garbage collection routine that makes it hang up if it has to try to relocate string arrays. Unless you try to do some fancy string array manipulations in big loops, you probably won't run into trouble. The FRE(x) routine at AFAD calls the garbage collector before finding out how much room is left between (81,82) and (7f,80)--in case you wnat to go bug hunting.

NUMERIC VARIABLE REPRESENTATION

The floating point value of a numeric variable is stored in its four bytes in normalized binary exponential (scientific) nototion:



This would be read as: $.101_2 \times 2_{10}^{3} = 5_{10}$

The last three bytes contain the number, to 24 bits' accuracy. The first byte is the power of 2--if you like, the number of places to move the binary point. (The binary point is like the decimal point, except to its right we have the ½'s column, ¼'s column, 1/8's column, etc--instead of 1/10's, 1/100's, etc.)

The most significant bit of the value (bit 7 of byte 2) is always interpreted as having the value 1. (If it were 0, we could shift the number to the left (binary point to the right) until it was 1, increasing the exponent by as many places as we moved.) Since this is understood, we can use that actual bit in memory as the sign bit. (1 is negative) Negative numbers are not represented in 2's complement form. The exponent, however, is. Some examples:

5	10000011	00100000	00000000	00000000
1	10000001	00000000	00000000	00000000
2	10000010	00000000	00000000	00000000
3	10000010	01000000	00000000	00000000
4	10000011	00000000	00000000	00000000
7	10000011	01100000	00000000	00000000
15	10000100	01110000	00000000	00000000
-5	10000011	10100000	00000000	00000000
(3/8).375	01111111	01000000	00000000	00000000
0	00000000	00000000	00000000	00000000

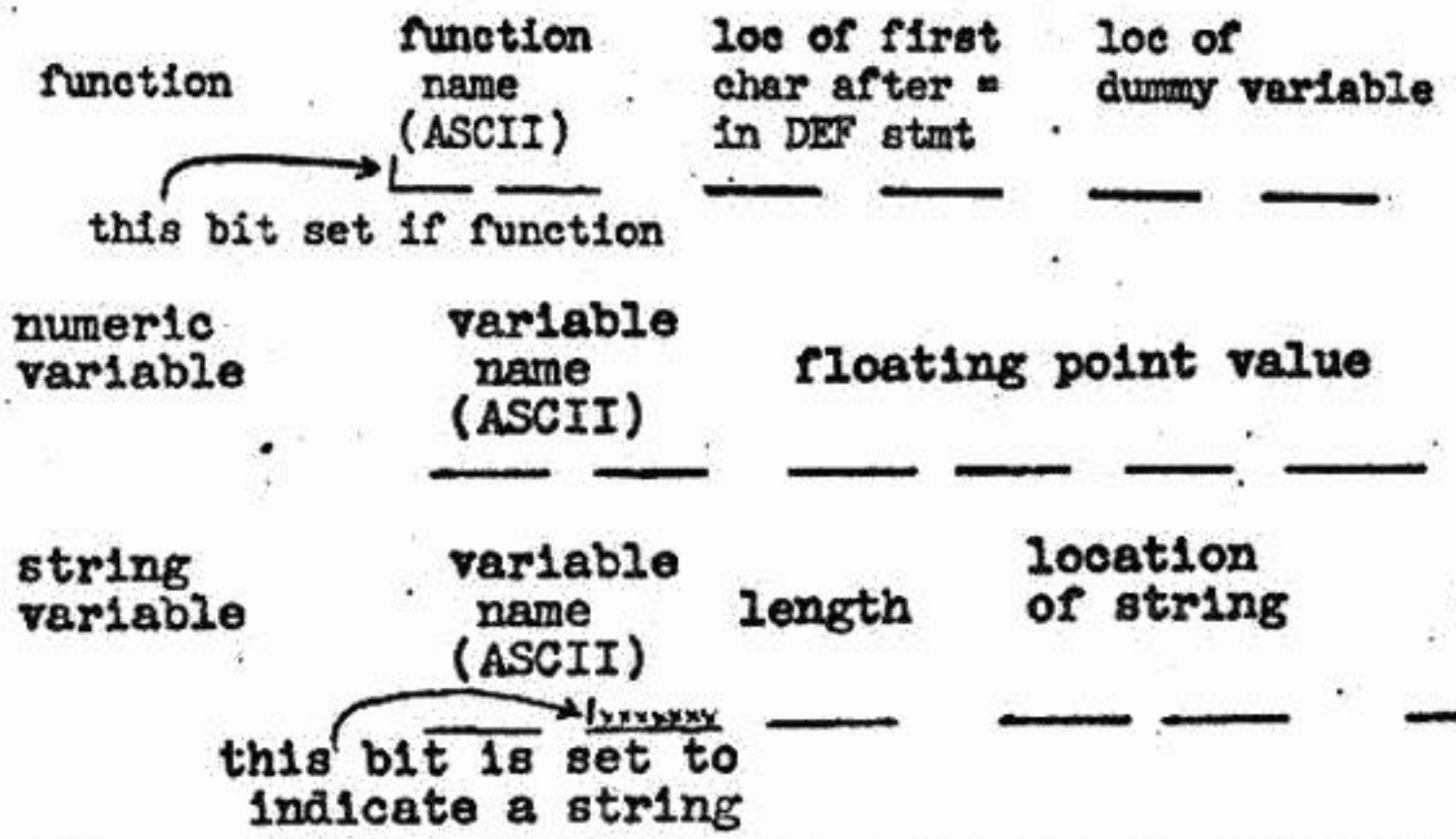
If you want to explore this further, there follows a short basic program to read the binary representation of a number from memory. It looks at the 2nd thru 4th bytes after (7B,7C). Killing line 30 lets you look at the variable name (and the first two bytes of the value).

If you also replace 7B,7C, programs are editable and can run happily. NOTE: Either avoid programs with lots of variables that can wipe out other programs, or also update 85,86 to indicate that the top of memory is just below the next program up. The hard one to fix is 7B,7C. It points to variable workspace--so BASIC POKE statements using variables can't fix it: the variables are lost between the first and second POKEs!

BASIC VARIABLE STORAGE

BASIC also needs space to store variables. These are stored in memory above the program--numeric variables, preceeded by their names from the end of the program going up, and string variables from the top of memory going down--their names being kept in a table along with where in memory the strings actually lime. Two data areas(with name tables) are kept--one for arrays (string and numeric), the other for single variables (string or not) and functions. Since only 7 bits are needed for each character of the variable name, the highest bits are used to show what type of variable is stored. A 1 in the second character indicates a string. A 1 in the first character indicates a function. (In DEF FNAB(X).) Both first bits high indicates a string function (FNAB$), although the system does not support them.

Single variables are stored immediately following the program, starting at the address pointed at by 7B,7C on page zero. (The abbreviation (7B,7C) is used to indicate the contents of 7B,7C. Thus, the sihgle variables start at (7B,7C).) Each variable is stored in a(fixed length) six byte block in this area:



To find a variable, BASIC searches the names, starting at (7B,7C), skipping to the next name 6 bytes later'til a match is found.(If a string is being searched for, the actual string is not here, but at the address contained in the 4th and 5th bytes.) The search ends if a match is not found by the end of the area, (7D,7E).

TOKENS AND BASIC STORAGE

Your BASIC programs are stored, line by line, in a partially pre-digested form starting(normally) at memory location 0301. All BASIC keywords (FOR, GOTO, END, =, CHR$, etc.) are stored as one-byte "tokens". Tokens always have the highest bit on (i.e., they are always greater than 128_{10}.) Other parts of your BASIC statements (like AA and 123 in LET AA=123) are stored as the ASCII characters you typed in. The line number is stored as a two-byte straight binary number. (That does not explain why the highest allowed line number is 63999 instead of 65535!) In addition to these, each stored line of BASIC source contains a two byte pointer containing the address of the next stored BASIC line. (This lets BASIC search rapidly for a given line number.) The format of BASIC statement storage is always like this:

null pointer to next line line # BASIC code; tokens and ASCII null of next line

(That information alone is enough to let you write a BASIC program renumbering program.)

The "normally starting at 0301" can provide interesting possibilit "BASIC workspace"--the area in memory where your program and variables are stored--begins at whatever address is contained in locations 0079,007A. (Machine addresses arenormally stored lo byte, hi byte. Thus, when the coldstart routine initializes these locations, it puts 01 in 0079 and 03 in 007A.) Now, if you change this (with your trusty ROM monitor or with POKE statements), you can make BASIC store your programsanywhere you choose. In fact, you could have one program stored starting at 0301, another at 0901, and another... all using the same line numbers, if you want! BASIC will find only one at a time for running and listing--the one whose beginning is contained in 79,7A. Note: the byte immediately before the first line must be the initial null. Normally, the system puts a permanent 0 in loc 0300, and the first byte of the first pointer goes in 301. You must put the initial null in (at 0900 in the example above) or nothing works. After you change 79,7A and put in that initial zero, type NEW to reset some other pointers. Unfortunately, if you put one program one place, reset 79,7A and put another somewhere else, trying to edit the first one will blow up the second program and not work in the first. You can, however switch back and forth if all you do is run and list the programs. ~~(A little fancy work with~~

One significant fact shows that more than one person worked on MICROSOFT who did not tell the other guy what he was doing. The convention for storing a string function varies. Names of string variables store the $ after the name and some string functions store the $ in the text. Other string functions such as MID$ store only on the one byte command and assume the presence of the $. Look for it either way in the text if you are looking for a string variable. Brackets show the same inconsistency. Some functions which require operators store both brackets, but other store only the second bracket, ")", and assume the presence of the first. That doesn't mean that you can leave them out when you type in the text, just that you can't find them if you look at the stored code.

That actually ends the line of MICROSOFT, but while we are at it, we might as well look at the stuff that follows. The first thing is the variable tables. The variables are stored in the order that they are found in the text. (For fast access, initialize the most used variable first.) Each variable takes six bytes. Regular variables start with two bytes for the name (even if it is a one byte name), followed by three bytes of packed BCD giving the six digit value and ending with a 0 which marks the end of the table entry.

String variables are stored - sort of - in the same table. Their entries are also six bytes long. The differences start in the second ASCII character of the string name plus 128 to show that it is a string entry. The next bytes store the address of the last place that it appeared in the text. The end is marked by 0.

If you look at the sample program, you will notice that the first thing following the 0 that ends the program is 66 0. The ASCII representation of the variable name "B". If it were a two letter name, the second bytes would contain the second character of the name. The next three bytes contain the six digit value of "B" in packed BCD, which is a bear to read. The last zero marks the end of the table entry. The next entry (65 128 1 13 3 0) contains the information on A$. The two byte name is coded with ASCII for the letter and the ASCII plus 128 for the next letter to show that it is a string. The next byte is the length of the string and the next two bytes (13 3) are the memory location where the first text reference is found. The 0 marks the end of the table. The last entry in the table is for variable X which shows the current value of 831 (88 0 138 78 128 0).

If there were subscripted variables, they would follow the regular variable table (to look at a subscripted variable table, change line 10 to 10 A(1)=1:A(2)=31 and change 830 in line 20 to 800. The table begins with a variable name, the length of the table and then the actual entries. The system marks off four spaces for each entry - one for the marker number and three for the six digits of the variable value. It sets aside enough space

HANDY LOCATIONS IN ROM BASIC

PAGE 0

0000 JUMP TO WARM START (4C/74/A2)
00FB CASSETTE/KEYBOARD FLAG
00FC DATA TEMPORARY HOLD FOR MONITOR

PAGE 1

0100-0141 STACK
0130 NMI VECTOR. NMI INTERRUPT CAUSES A JUMP TO THIS LOCATION
01C0 IRQ VECTOR

PAGE 2

0200 CURSOR POSITION
0203 LOAD FLAG
0204 SAVE FLAG
0206 CRT SIMULATOR BAUD RATE-VARIES FROM 0=FAST to FF=SLOW BAUD RATE
0212 CONTROL C FLAG
0218 INPUT VECTOR (C1P only)
021A OUTPUT VECTOR
021C CONTROL C CHECK VECTOR
021E LOAD VECTOR
0220 SAVE VECTOR
0222-022FA **UNUSED** A NICE PLACE TO PUT USR ROUTINES

PAGE 3 and up to end of RAM is BASIC work space.

A000-BFFF BASIC IN ROM
D000-D3BF VIDEO REFRESH MEMORY
DF00 POLLED KEYBOARD
F000-F001 CASSETTE PORT ACIA (C1P)
F800-FFFF MONITOR EPROM
FC00 FLOPPY BOOTSTRAP
FD00 KEYBOARD INPUT ROUTINE (SEE "INPUTTING WITHOUT SCROLLS")
FF00 BASIC I/O SUPPORT

USEFUL SUBROUTINES IN ROM

A274 BASIC warm start *NOTE-FOR DISK BASIC WARM START IS 051A*
BD11 BASIC cold start
BF2D CRT simulator-prints character in Accumulator to screen offset by value in 0200
FD00 Input character from keyboard result in A and in 0213
FCB1 Output character in A to cassette
FE00 Entry to Monitor-
FE00 Entry to Monitor -bypass stack initialization.
FE93 Converts ASCII hex to binary-result in A.-80 if bad value
FF69 BASIC output to cassette routine-outputs one character to port and screen, outputs 10 nulls if character is a carriage return.
FFBA BASIC input routine
FF9B Control C routine
FF00 Reset entry point

SEMI FAST SCREEN CLEAR (WITHOUT THE USR FUNCTION)

I hate to be bothered with the USR screen clear. I can't remember it off hand and I hate to take time to look it up. Besides, it takes too much memory. This one is fast-it clears the screen in less than 2.16 seconds-and easier to remember

C2/4/8

```
100FORX=1TO29:?:NEXT
110FORX=55168TO55295:POKEX,32:NEXT
```

C1P

```
100FORX=1TO29:?:NEXT
110FORX=54174TO54275 (54307 on some
   monitors):POKEX,32:NEXT
```

PRINT AT STATEMENT

OSI has a great BASIC but the lack of a PRINT AT command makes it difficult to print scores and names and similar items where you want them on the screen. You usually end up with a long series of POKE statements and you have to divide the score up into individual digits to do even that. There is a simple solution. Add this subroutine to your program-

```
5000FORY=1TOLEN(D$):POKED+Y,ASC(MID$(D$,Y,1)):NEXT:RETURN
```

To POKE up any name, word, or even sentence on the screen simply set the name equal to D$ and make D=equal the starting address on the screen. i.e.

```
300D$="WINNER IS":D=54040:GOSUB5000
```

Scores should be done just a little differently. You start at the second digit because the BASIC thinks the sign is the first digit in the string and can set you over one space from where you planned. You may also want to blank the digit after the string to allow for the possibility that the score may decrease (say from three to two digits). To use it you set the score equal to D$ and the final product looks like this-

```
300D$=STR$(SCORE):D=54040:GOSUB5000
5000FORY=2TOLEN(D$):POKED+Y,ASC(MID$(D$,Y,1)):NEXT
5010POKED+Y,32:RETURN
```

SOME POKES YOU SHOULD KNOW

To aid in reading you may want to set the line length down to 32 on a C2 or to 23 on a C1. Unfortunately, if you set them down when you start up the system you will be unable to make tapes. Fortunately. the line length is stored in location 15. You can reset line length by executing 100POKE15,32 (or any other number down to as little as one) and then reset with 200POKE15,72 to record the program.

If you find it annoying to reserve space for user programs when you fire up the system (I always forget to do it when I am using the rapid screen clear) you can set the memory space by POKEing the high order digit (in HEX) into location 134 and the low order digit into 133. For instance, the line 100POKE134,14 will reserve space for the screen clear without reseting the system.

You can even make self starting BASIC programs if you are willing to do a few additional moments work when you make the tape. The flag for LOAD is in location 515. A 1 POKEd into that location turns off the load mode. Therefor, to make a self start tape-as soon as the program finishes reading out to the tape and while the system is still in SAVE mode, type in POKE515,1:RUN

That command will record on the tape and start the program automatically when it finishes loading.

SAVE can be turned off in a similar manner by POKEing a 0 into location 517

EASY KEY DETECTION

If you are doing a one player game, you can detect the control keys without either POKEing the keyboard or turning off the CONTROL C scan. The values for the shifts, rept, control, and esc keys are recorded continuously in location 57100. i.e. If you push the right shift, a 3 always appears in 57100. To see how it works try this program

```
10PRINTPEEK(57100):GOTO10
```

Then push the control keys one at a time. It is simple, fast, and allows you to keep the CONTROL C function to break the program..

COVER ART BY TULLIO PRONI